

Eşzamanlılık: Giriş

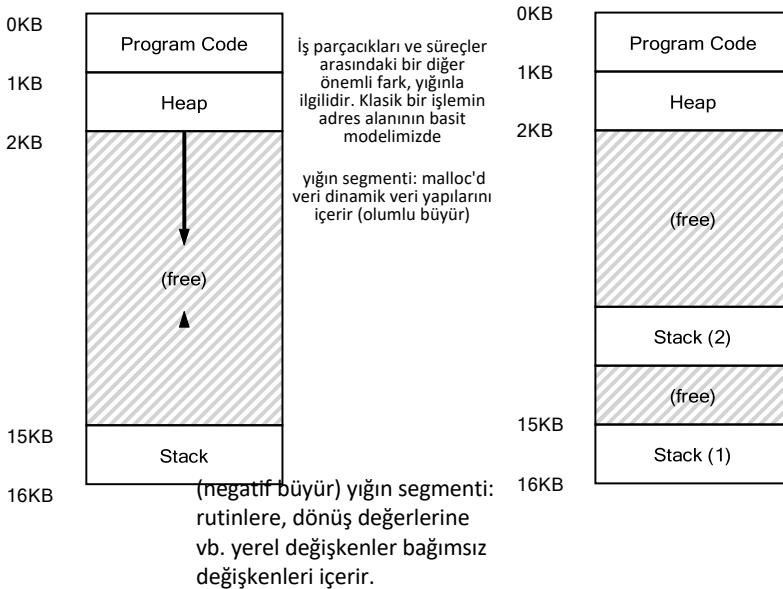
Şimdiye kadar, işletim sisteminin gerçekleştirdiği temel soyutlamaların gelişimini gördük. Tek bir fiziksel CPU'nun nasıl alınacağını ve birden fazla **sanal CPU'ya(virtual CPU)** nasıl dönüştürüleceğini gördük, böylece aynı anda çalışan birden fazla program yanısamasını sağladık. Ayrıca, her işlem için büyük, özel bir **sanal bellek(virtual memory)** yanısamasının nasıl yaratılacağını da gördük; **Adres alanının(address space)** bu şekilde kısıtlanması, işletim sistemi gerçekten de fiziksel bellekteki (ve bazen disk) adres alanlarını gizlice çoğaltlığında, her programın kendi belleğine sahipmiş gibi davranışmasını sağlar.

Bu notta, çalışan tek bir süreç için yeni bir soyutlama sunuyoruz: **bir iş parçacığınınıktır(thread)** Bir program içindeki tek bir yürütme noktasına (yani, talimatların getirildiği ve yürütüldüğü tek bir bilgisayar) ilişkin klasik görüşümüz yerine, çok iş parçacıklı bir programın **birden fazla yürütme noktası(multi-threaded)** vardır (yani, her biri getirilen ve yürütülen birden fazla bilgisayar). Belki de bunu düşünmenin başka bir yolu, bir fark dışında, her iş parçacığının ayrı bir işleme çok benzemesidir: aynı adres alanını paylaşırlar ve böylece aynı verilere erişebilirler.

Bu nedenle, tek bir iş parçacığının durumu, bir işleminkine çok benzer. Programın talimatları nereden aldığı izleyen bir program sayacı (PC) vardır. Her iş parçacığının hesaplama için kullandığı kendi özel kayıt kümesi vardır; Bu nedenle, tek bir işlemcide çalışan iki iş parçacığı varsa, birini (T1) çalıştmaktan diğerini (T2) çalışmaya geçerken, **bağlam(context switch)** değiştirme işlemi yapılmalıdır. İş parçacıkları arasındaki bağlam geçisi, işlemler arasındaki bağlam geçisine oldukça benzer, çünkü T1'in kayıt durumu kaydedilmeli ve T2'nin kayıt durumu T2'yi çalıştırmadan önce geri yüklenmelidir.

İşlemlerle, durumu bir **süreç kontrol bloğuna(process control block (PCB))** kaydettik; artık, bir işlemin her iş parçacığının durumunu depolamak için bir veya daha fazla **iş parçacığı kontrol bloğuna(thread control blocks (TCB))** ihtiyacımız olacak. Bununla birlikte, işlemlere kıyasla iş parçacıkları arasında gerçekleştirdiğimiz bağlam anahtarlarında önemli bir fark vardır: adres alanı aynı kalır (yani, hangi sayfa tablosunu kullandığımızı değiştirmeye gerek yoktur).

İş parçacıkları ve süreçler arasındaki bir diğer önemli fark, yığınla ilgilidir. Klasik bir işlemin adres alanının basit modelimizde 1



Şekil 26.1: Tek İş Parçacıklı ve Çok İş Parçacıklı Adres Alanları (**Single-Threaded And Multi-Threaded Address Spaces**)

(artık tek iş parçacıklı işlem(**single-threaded**) diyebiliriz), genellikle adres alanının altında bulunan tek bir yiğin vardır (Şekil 26.1, solda).

Bununla birlikte, çok iş parçacıklı bir süreçte, her iş parçacığı bağımsız olarak çalışır ve elbette yaptığı işi yapmak için çeşitli rutinlere çağrılabılır. Adres alanında tek bir yiğin yerine, iş parçacığı başına bir tane olacaktır. İçinde iki iş parçacığı bulunan çok iş parçacıklı bir sürecimiz olduğunu varsayıyalım; ortaya çıkan adres alanı farklı görünüyor (Şekil 26.1, sağda).

Bu şekilde, işlemin adres alanı boyunca yayılmış iki yiğin görebilirsiniz. Böylece, yiğina ayrılan değişkenler, parametreler, yeniden dönüş değerleri ve yiğine koyduğumuz diğer şeyler, bazen **iş parçacığı yerel depolama(thread-local)** olarak adlandırılan şeye, yani ilgili iş parçacığının yiğinına yerleştirilecektir.

Bunun güzel adres alanı düzenimizi nasıl mahvettiğini de fark edebilirsiniz. Önceden, yiğin ve yiğin bağımsız olarak büyüyebiliyordu ve sorun yalnızca adres alanında yeriniz bittiğinde ortaya çıkıyordu. Burada artık böyle güzel bir durumumuz yok. Neyse ki, yiğinların genellikle çok büyük olması gerekmemişinden (istisna, özyinelemenin yoğun bir şekilde kullanıldığı programlarda) bu genellikle sorun değildir.

26.1 Neden Threads Kullanmalı?

İş parçacıklarının ayrıntılarına ve çok iş parçaklı programlar yazarken karşılaşabileceğiniz bazı sorunlara girmeden önce, önce daha basit bir soruyu cevaplayalım. Neden iplik kullanmalısınız?

```
1 #include <stdio.h>
2 #include <assert.h>
3 #include <pthread.h>
4 #include "common.h"
5 #include "common_threads.h"
6
7 void *mythread(void *arg) {
8     printf("%s\n", (char *) arg);
9     return NULL;
10 }
11
12 int
13 main(int argc, char *argv[]) {
14     pthread_t p1, p2;
15     int rc;
16     printf("main: begin\n");
17     Pthread_create(&p1, NULL, mythread, "A");
18     Pthread_create(&p2, NULL, mythread, "B");
19     // katılan iş parçacıklarının bitmesini bekler
20     Pthread_join(p1, NULL);
21     Pthread_join(p2, NULL);
22     printf("main: end\n");
23     return 0;
24 }
```

Şekil 26.2: Basit İş Parçacığı Oluşturma Kodu (t0.c)

Görünüşe göre, iplikleri kullanmanız için en az iki ana neden var. Birincisi basit: **parallelilik(parallelism)**. Çok büyük diziler üzerinde işlem gerçekleştiren bir program yazdığınıza düşünün, örneğin, iki büyük diziyi birbirine ekleyerek veya dizideki her ögenin değerini bir miktar artırarak. Sadece tek bir prosedür üzerinde çalışıyorsanız, görev basittir: sadece her işlemi gerçekleştirin ve yapın. Bununla birlikte, programı birden çok işlemcili bir sisteme yürütüyorsanız, her biri işin bir bölümünü gerçekleştirmek için işlemcileri kullanarak bu işlemi önemli ölçüde hızlandırma potansiyeline sahipsiniz. Standart **tek iş parçacıklı(single-threaded)** programınızı birden çok CPU üzerinde bu tür bir iş yapan bir programa dönüştürme görevine **parallelleştirme(parallelization)** denir ve bu işi yapmak için CPU başına bir iş parçacığı kullanmak, programların modern donanımında daha hızlı çalışmasını sağlamanın doğal ve tipik bir yoludur.

İkinci neden biraz daha inceliklidir: yavaş olması nedeniyle program ilerlemesini engellemekten kaçınmak I/O. Farklı I/O türlerini gerçekleştiren bir program yazdığınıza düşünün: açık bir disk I/O'sinin tamamlanması için bir bilge göndermeye veya almayı bekliyorsunuz., veya hatta (örtük olarak) bir sayfa hatasının bitmesi için. Beklemek yerine, programınız hesaplama yapmak için CPU'yu kullanmak veya hatta daha fazla I/O isteği göndermek de dahil olmak üzere başka bir şey yapmak isteyebilir. İplikler kullanmak, sıkışık kalmamak için doğal bir yoldur; programınızdaki bir iş parçacığı beklerken (yani, I/O beklerken engellenir), CPU zamanlayıcısı çalışmaya ve yararlı bir şey yapmaya hazır olan diğer iş parçacıklarına geçebilir. İş parçacığı oluşturma, I/O'nin tek bir programdaki diğer etkinliklerle **örtüşmesini(overlap)** sağlar, tipki **çoklu programlama(multiprogramming)** programların programlar arasındaki işlemler için yaptığı gibi; Sonuç olarak, birçok modern sunucu tabanlı uygulama (web sunucuları, veritabanı yönetim sistemleri ve benzeri) uygulamalarında iş parçacıklarından yararlanır.

Tabii ki, yukarıda belirtilen durumlardan herhangi birinde, iş parçacıkları yerine çoklu işlemler kullanabilirsiniz. Ancak, iş parçacıkları bir adres alanını paylaşır ve böylece veri paylaşımını kolaylaştırır ve bu nedenle bu tür programları oluştururken doğal bir seçimdir. Süreçler, veri yapılarının bellek olarak çok az paylaşılmasının gerekliliği mantıksal olarak ayrı görevler için daha sağlam bir seçimdir.

26.2 Bir Örnek: İş Parçacığı Oluşturma

Bazı ayrıntılara girelim. Diyelim ki her biri bağımsız bir çalışma yapan iki iş parçacığı oluşturan bir program çalıştırmak istedik, bu durumda "A" veya "B" yazdırıyoruz. Kod Şekil 26.2'de gösterilmiştir (sayfa 4).

Ana program, her biri `mythread()` işlevini çalıştıracak iki iş parçacığı oluşturur, ancak farklı bağımsız değişkenlerle (A veya B dizesi). Bir iş parçacığı oluşturulduktan sonra, hemen çalışmaya başlayabilir (zamanlayıcının kaprislerine bağlı olarak); alternatif olarak, "hazır" ancak "çalışan" bir duruma getirilmeyebilir ve bu nedenle henüz çalıştırılmayabilir. Tabii ki, çok işlemcili bir işlemcide, iş parçacıkları aynı anda çalışıyor bile olabilir, ancak endişelenmemeyelim. Bu olasılık hakkında henüz.

Bu küçük programın olası yürütme düzenini inceleyelim. Yürütme diyagramında (Şekil 26.3, sayfa 5), zaman aşağı yönde artar ve her sütun farklı bir iş parçacığının (ana iş parçacığı veya İş Parçacığı 1 veya İş Parçacığı 2) ne zaman çalıştığını gösterir.

Bununla birlikte, bu siparişin mümkün olan tek sipariş olmadığını unutmayın. Aslında, bir dizi talimat verildiğinde, zamanlayıcının belirli bir noktada hangi iş parçacığını çalıştırımıya karar verdiğine bağlı olarak oldukça az sayıda vardır. Örneğin, bir iş parçacığı oluşturulduktan sonra hemen çalışabilir ve bu da Şekil 26.4'te (sayfa 5) gösterilen yürütmeye yol açar.

Ayrıca, "B"nin "A"dan önce yazdırıldığını bile görebiliriz, örneğin, zamanlayıcı Thread 1 daha önce oluşturulmuş olmasına rağmen önce Thread 2'yi çalıştırırmaya karar verdiyse; ilk oluşturulan bir iş parçacığının önce çalışacağını varsayılmak için hiçbir neden yoktur. Şekil 26.5 (sayfa 6) bu son yürütme sırasını gösterir ve Thread 2, Thread 1'den önce eşyalarını destekler.

Görebileceğiniz gibi, iş parçacığı oluşturma hakkında düşünmenin bir yolu

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
	runs	
	prints "B"	
	returns	
prints "main: end"		

Şekil 26.3: İş Parçası İzleme (1)

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
	runs	
	prints "B"	
	returns	
waits for T1		

returns immediately; T1 is done

waits for T2

returns immediately; T2 is done

prints "main: end"

Şekil 26.4: İplik İzleme (2)

bir fonksiyon çağrısı yapmak gibi bir şey olmasıdır; ancak, önce işlevi kesip sonra araya geri dönmek yerine, sistem bunun yerine çağrılan rutin için yeni bir yürütme iş parçası oluşturur ve arayandan bağımsız olarak, belki de yaratılmadan dönmeden önce, ancak belki de çok daha sonra çalışır. Daha sonra ne çalışacağı işletim sistemi zamanlayıcısı tarafından belirlenir ve zamanlayıcı muhtemelen mantıklı bir algoritma uygulasa da, herhangi bir anda neyin çalışacağını bilmek zordur.

Bu örnekten de anlayabileceğiniz gibi, iplikler hayatı karmaşıklaştırır: neyin ne zaman çalışacağını söylemek zaten zor! Bilgisayarlar eşzamanlılık olmadan anlamak için yeterince zordur. Ne yazık ki, eşzamanlılık ile, sadece daha da kötüleşir. Çok daha kötüsü.

main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
	runs	
	prints "B"	
	returns	
bekler(waits) for T1	runs	
	prints "A"	
	returns	
bekler(waits) for T2		
hemen geri döner; T2 baskıları		
bitti "main: end"		

Figure 26.5: İplik İzleme(Thread Trace) (3)

26.3 Neden Daha da Kötüye Gidiyor: Paylaşılan Veriler

Yukarıda gösterdiğimiz basit iş parçacığı örneği, iş parçacıklarının nasıl oluşturulduğunu ve zamanlayıcının bunları çalıştırma nasıl karar verdiğine bağlı olarak farklı sıralarda nasıl çalışabileceklerini göstermede yararlı oldu. Ancak size göstermediği şey, iş parçacıklarının paylaşılan verilere erişiklerinde nasıl etkileşimde bulunduklarıdır.

İki iş parçacığının genel paylaşılan bir değişkeni güncelleştirmek istediği basit bir örnek düşünelim. İnceleyeceğimiz kod Şekil 26.6'dadır (sayfa 7).

İşte kod hakkında birkaç not. İlk olarak, Stevens'in önerdiği gibi [SR05], iş parçacığı oluşturmayı sariyoruz ve başarısızlıktan çıkmak için rutinleri birleştiriyoruz; Bu kadar basit bir program için, en azından bir hata olduğunu fark etmek istiyoruz (eğer öyleyse), ancak bu konuda çok akıllı bir şe yapsamamak istiyoruz. O (e.g., just exit). Thus, Pthread create() simply calls pthread create() ve dönüş kodunun 0; değilse, Pthread create() sadece bir mesajı yazdırır ve çıkar.

İkincisi, çalışan iş parçacıkları için iki ayrı işlev gövdesi kullanmak yerine, yalnızca tek bir kod parçası kullanırız ve iş parçacığına bir tartaşma (bu durumda bir dize) geçiririz, böylece her iş parçacığının iletisilerinden önce farklı bir harf yazdırmasını sağlayabiliriz.

Son olarak ve en önemlisi, şimdi her çalışanın ne yapmaya çalıştığını bakabiliriz: paylaşılan değişkene bir sayı ekleyin counter, ve bunu yapın 10 milyon kez (1e7) bir döngü içinde. Böylece, istenen nihai sonuç: 20,000,000.

Şimdi nasıl davranışını görmek için programı derleyip çalıştırıyoruz. Bazen, her şe y bekleyebileceğimiz gibi çalışır:

```
prompt> gcc -o main main.c -Wall -pthread; ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include "common.h"
4 #include "common_threads.h"
5
6 static volatile int counter = 0;
7
8 // mythread()
9 //
10 // Bir döngüde tekrar tekrar sayaç için 1 ekler
11 // Hayır, 10.000.000'u bu şekilde eklemezsiniz.
12 // bir sayaç, ama sorunu güzel bir şekilde
13 // gösteriyor.
14 //
15 void *mythread(void *arg) {
16     printf("%s: begin\n", (char *) arg);
17     int i;
18     for (i = 0; i < 1e7; i++) {
19         counter = counter + 1;
20     }
21     printf("%s: done\n", (char *) arg);
22     return NULL;
23 }
24
25 // main()
26 //
27 // Sadece iki iş parçacığı başlatır (pthread_create)
28 // ve sonra onları bekler (pthread_join)
29
30 int main(int argc, char *argv[]) {
31     pthread_t p1, p2;
32     printf("main: begin (counter = %d)\n", counter);
33     Pthread_create(&p1, NULL, mythread, "A");
34     Pthread_create(&p2, NULL, mythread, "B");
35
36     // katılan iş parçacıklarının bitmesini bekler
37     Pthread_join(p1, NULL);
38     Pthread_join(p2, NULL);
39     printf("main: done with both (counter = %d)\n",
40           counter);
41     return 0;
42 }
```

Figure 26.6: Veri Paylaşımı: Uh Oh (t1.c)

Ne yazık ki, bu kodu çalıştırduğumızda, tek bir işlemcide bile, mutlaka istenen sonucu elde edemeyiz. Bazen şunları elde ederiz:

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Bir kez daha deneyelim, sadece delirip delirmemiizi görmek için. Sonuçta, bilgisayarların **deterministic(deterministic)** sonuçlar üretmesi gerekmiyor mu?, size öğretildiği gibi?! Perhaps your professors have been lying to you? (*gasp*)

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Her çalışma sadece yanlış olmakla kalmaz, aynı zamanda farklı bir sonuç verir! Büyük bir soru kaldı: bu neden oluyor??

İpucu: ARAÇLARINIZI BILIN VE KULLANIN

Her zaman yazmanıza yardımcı olacak yeni araçlar öğrenmelisiniz, hata ayıklama(debug), ve un-derstand bilgisayar sistemleri. Burada, **sökücü(disassembler)** adı verilen düzgün bir araç kullanıyoruz. Bir yürütülebilir dosyada bir çözücü çalıştırığınızda, programı hangi montaj talimatlarının oluşturduğunu gösterir. Mesela, anlamak istiyorsak düşük düzeyli(low-level) sayacı güncelleştirmek için kod(örneğimizde olduğu gibi), koşuyoruz objdump (Linux) Derleme kodunu görmek için:

```
prompt> objdump -d main
Bunu yapmak, programdaki tüm talimatların düzgün bir şekilde etiketlenmiş uzun bir listesini oluşturur (özellikle -g bayrak(flag)), programdaki sembol bilgilerini geçersiz kılar. Burası objdump Program, nasıl kullanılacağını öğrenmeniz gereken birçok araçtan sadece biridir.; bir debugger gibi gdb, bellek profil oluşturucular gibi valgrind veya purify, ve elbette derleyicinin kendisi, hakkında daha fazla bilgi edinmek için zaman harcamanız gereken diğerleridir.; aletlerinizi ne kadar iyi kullanırsanız, o kadar iyi sistemler kurabilirsiniz.
```

26.4 Sorunun Kalbi: Kontrolsüz Zamanlama

Bunun neden olduğunu anlamak için, derleyicinin güncelleme için oluşturduğu kod benzerliğini anlamalıyız. counter. Bu durumda, sadece bir sayı (1) eklemek istiyoruz. counter. Bu nedenle, bunu yapmak için kod dizisi şöyle bir şeye benzeyebilir. (in x86);

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Bu örnek, değişken sayacın adres 0x8049a1c bulunduğuunu varsayar. Bu üç talimatı dizide, x86 mov talimatı ilk önce adresteki bellek değerini almak ve bunu kayıt eax'ına koymak için kullanılır. Ardından, eax kaydının içeriğine 1 (0x1) eklenecek ekleme gerçekleştirilir ve son olarak, eax içeriği aynı adreste belleğe geri depolanır.

İki iş parçacığımızdan birinin (İş Parçacığı 1) kodun bu bölgesine girdiğini ve böylece sayacı bir kat artırmak üzere olduğunu düşünelim. Sayacın değerini (diyelim ki başlamak için 50'dir) kayıt eax'ına yükler. Böylece, Thread 1 için eax=50. Sonra kayıt defterine bir tane ekler; böylece eax=51. Şimdi, talihsiz bir şey olur: bir zamanlayıcı kesintisi söner; Böylece, işletim sistemi çalışmaktı olan iş parçacığının durumunu (PC'si, eax dahil yazmaçları vb.) iş parçacığının TCB'sine kaydeder.

Şimdi daha kötü bir şey olur: Thread 2 çalıştırılmak için seçilir ve aynı kod parçasını içerir. Ayrıca, sayacın değerini alarak ve eax'ına koyarak ilk talimatı uygular (unutmayın: çalışırken her iş parçacığının kendi özel kayıtları vardır; kayıtlar, onları kaydeden ve geri yükleyen bağlam anahtarı kodu tarafından sanallaştırılır). Bu noktada sayacın değeri hala 50'dir ve bu nedenle Thread 2 eax=50'ye sahiptir. Daha sonra Thread 2'nin sonraki iki yönergeyi yürüttüğünü, eax'ı 1 artırlığını (dolayısıyla eax=51) ve ardından eax içeriğini sayaca (adres 0x8049a1c) kaydettiğini varsayıyalım. Böylece, global değişken sayacı artık 51 değerine sahiptir.

Son olarak, başka bir bağlam anahtarı oluşur ve İş Parçacığı 1 çalışmaya devam eder. Taşıma ve ekleme işlemini yeni gerçekleştirdiğini ve şimdi son taşıma talimatını yerine getirmek üzere olduğunu hatırlayın. Ayrıca eax = 51 olduğunu da hatırlayın. Böylece, son mov talimatı yürütülür ve değeri belleğe kaydeder; sayaç tekrar 51 olarak ayarlanır.

Basitçe söylemek gerekirse, olan şey şudur: sayaç artırma kodu iki kez çalıştırılmıştır, ancak 50'de başlayan sayaç şimdi yalnızca 51'e eşittir. Bu programın "doğru" bir sürümü, değişken sayacın 52'ye eşit olmasıyla sonuçlanmalıdır.

Sorunu daha iyi anlamak için ayrıntılı bir yürütme izlemesine bakalım. Bu örnek için, yukarıdaki kodun aşağıdaki sıra gibi bellekte 100 adresinde yüklediğini varsayıyalım (güzel, RISC benzeri komut kümelerine alışkin olanlarınız için not: x86'nın değişken uzunlukta talimatları vardır; bu mov talimatı 5 bayt bellek kaplar ve yalnızca 3 tane ekleyin):

OS	Thread 1	Thread 2	(after instruction)		
			PC	eax	counter
	<i>kritik bölümden önce</i>		100	0	50
	mov 8049a1c,%eax		105	50	50
	add \$0x1,%eax		108	51	50
Kesme(Interrupt)					
<i>Kayıt etmek(save) T1</i>					
<i>Eski haline getirmek(restore) T2</i>			100	0	50
			mov 8049a1c,%eax	105	50
			add \$0x1,%eax	108	51
			mov %eax,8049a1c	113	51
interrupt					
<i>save T2</i>					
<i>restore T1</i>			108	51	51
			mov %eax,8049a1c	113	51

Şekil 26.7: Sorun: Yakın ve Kişisel

```

100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c

```

Bu varsayımlarla, neler olduğu Şekil 26.7'de gösterilmiştir (sayfa 10). Sayacın 50 değerinden başladığını varsayıyalım ve neler olup bittiğini anladığınızdan emin olmak için bu örneği izleyin.

Burada gösterdiğimiz şey **yarış durumu(race condition)** (veya daha spesifik olarak bir **veri yarışı(data race)**) olarak adlandırılır: sonuçlar, kodun zamanlama uygulamasına bağlıdır. Bazı kötü şanslarla (yani, yürütmenin zamanında olmayan noktalarda meydana gelen bağlam anahtarları), yanlış sonuç alırız. Aslında her seferinde farklı bir sonuç alabiliriz; bu nedenle, güzel bir **deterministik(deterministic)** hesaplama yerine (bilgisayarlardan alışık olduğumuz), çıktıının ne olacağı **bilinmediği(indeterminate)** ve gerçekten de koşular arasında farklı olması muhtemel olan bu sonuca belirsiz diyoruz.

Bu kodu yürüten birden çok iş parçacığı bir yarış koşulunda neden olabileceğiinden, bu kodu kritik bir bölüm olarak adlandırıyoruz. **Kritik bölüm(critical section)**, paylaşılan bir değişkene (veya daha genel olarak paylaşılan bir kaynağa) erişen ve aynı anda birden fazla iş parçacığı tarafından yürütülmemesi gereken bir kod parçasıdır. Bu kod için gerçekten istedigimiz şey, **karşılıklı dışlama(mutual exclusion)** dediğimiz şeydir. Bu özellik, bir iş parçacığı kritik bölümde yürütülüyorsa, diğerlerinin bunu yapmasını engelleneceğini garanti eder.

Bu arada, bu terimlerin neredeyse tamamı, alanında öncü olan ve gerçekten de bu ve diğer çalışmalar nedeniyle Turing Ödülü'nü kazanan Edsger Dijkstra tarafından icat edildi; Sorunun şartlısı derecede açık bir tanımı için 1968 tarihli "İşbirliği Sıralı Süreçler" [D68] başlıklı makalesine bakınız. Kitabın bu bölümünde Dijkstra hakkında daha fazla şey duyacağız.

İpucu: ATOMİK İŞLEMLERI KULLANIN

Atomik işlemler, bilgisayar mimarisinden, mutabakat koduna (burada incelediğimiz şey), dosya sistemlerine (yakında inceleyeceğimiz), veritabanı yönetim sistemlerine ve hatta dağıtılmış sistemlere [L+93] kadar bilgisayar sistemlerinin oluşturulmasında en güçlü temel tekniklerden biridir.

Bir dizi eylemi atomik hale getirmenin ardından fikir basitte "ya hep ya hiç" ifadesiyle ifade edilir; ya birlikte gruplandırmak istediğiniz tüm birleşimler gerçekleştmiş gibi görünümlü ya da hiçbir gerçekleşmemiş, arada hiçbir durum görünmemelidir. Bazen, birçok aksiyonun tek bir **atomic(atomic)** eylemde gruplandırılmasına, veritabanları ve **işlem(transaction)** işleme dünyasında çok ayrıntılı olarak ortaya konan bir fikir olan işlem denir [GR92].

Eşzamanlılığı keşfetme temamızda, kısa talimat dizilerini atomik yürütme bloklarına dönüştürmek için senkronizasyon ilkellerini kullanacağız, ancak atomisite fikri, göreceğimiz gibi, bundan çok daha büyük. Örneğin, dosya sistemleri, sistem hataları karşısında doğru çalışması için kritik olan disk üzerindeki durumlarını atomik olarak geçirmek için günlük kaydı veya yazma üzerine kopyalama gibi teknikleri kullanır. Bu mantıklı değilse, endişelenmeyein - gelecekteki bir bölümde olacak. Eşzamanlılığı keşfetme temamızda, kısa talimat dizilerini atomik yürütme bloklarına dönüştürmek için senkronizasyon ilkellerini kullanacağız, ancak atomisite fikri, göreceğimiz gibi, bundan çok daha büyük. Örneğin, dosya sistemleri, sistem hataları karşısında doğru çalışması için kritik olan disk üzerindeki durumlarını atomik olarak geçirmek için günlük kaydı veya yazma üzerine kopyalama gibi teknikleri kullanır. Bu mantıklı değilse, endişelenmeyein - gelecekteki bir bölümde olacak.

26.5 Atomiklik Dileği

Bu sorunu çözmenin bir yolu, tek bir adımda, tam olarak yapmamız gereken her şeyi yapan ve böylece zamansız bir kesinti olasılığını ortadan kaldırın daha güçlü talimatlara sahip olmaktadır. Örneğin, ya şuna benzeyen süper bir talimatımız olsaydı:

```
memory-add 0x8049a1c, $0x1
```

Bu önergenin bellek konumuna bir değer kattığını ve donanımın **atomik olarak(atomically)** yürütülmesini garanti ettiğini varsayıyalım; talimat yürütüldüğünde, güncelleme istenildiği gibi gerçekleştirir. Öğretimin ortasında birbirine bağlanamazdı, çünkü donanımdan aldığımız garanti tam olarak budur: bir kesinti meydana geldiğinde, ya talimat hiç çalışmamıştır ya da tamamlanmıştır; arada bir durum yoktur. Donanım güzel bir şey olabilir, değil mi?

Atomik olarak, bu bağlamda, bazen "hepsi ya da hiçbirini" olarak kabul ettiğimiz "bir birim olarak" anlamına gelir. İstediğimiz şey, üç talimat dizisini atomik olarak yürütmektir:

```
mov 0x8049a1c, %eax  
add $0x1, %eax  
mov %eax, 0x8049a1c
```

Dediğimiz gibi, bunu yapmak için tek bir talimatımız olsaydı, sadece bu talimatı verebilir ve yapılabildirdik. Ancak genel durumda, böyle bir talimatımız olmayacağı. Eşzamanlı bir B ağacı inşa ettiğimizi ve onu güncellemek istediğimizi hayal edin; donanımın "B ağacının atomik güncellemesi" talimatını desteklemesini gerçekten ister miydik? Muhtemelen hayır, en azından aklı başında bir talimat setinde.

Bu nedenle, bunun yerine yapacağımız şey, donanımdan, **kronizasyon ilkeleri(syn-chronization primitives)** dediğimiz şeyin genel bir kümесini oluşturabileceğimiz birkaç yararlı talimat istemektir. Bu donanım desteğini kullanarak, işletim sisteminden gelen bazı yardımıcılara birleştiğinde, kritik bölmelere senkronize ve kontrollü bir şekilde erişen çok iş parçacıklı kod oluşturabileceğiz ve böylece eşzamanlı yürütmenin zorlu doğasına rağmen doğru sonucu güvenilir bir şekilde üretebileceğiz. Oldukça harika, değil mi?

Kitabın bu bölümünde inceleyeceğimiz sorun budur. Bu harika ve zor bir sorundur ve zihninizi incitmeli (biraz). Eğer yapmazsa, o zaman anlamıyorsun! Başınız ağrıyanaya kadar çalışmaya devam edin; o zaman doğru yöne gittiğinizi anlarsınız. Bu noktada bir mola verin; başınızın çok fazla ağrısını istemiyoruz.

PÜF NOKTA: SENKRONIZASYON NASIL DESTEKLENİR

Kullanıcıları senkronizasyon ilkeleri oluşturmak için donanımdan hangi desteği ihtiyacımız var? İşletim sisteminden hangi desteği ihtiyacımız var? Bu ilkeleri nasıl doğru ve verimli bir şekilde inşa edebiliriz? Programlar istenen sonuçları elde etmek için bunları nasıl kullanabilir?

26.6 Bir Sorun Daha Var: Bir Başkasını Beklemek

Bu bölüm, eşzamanlılık sorununu, iş parçacıkları arasında yalnızca bir etkileşim türü gerçekleşiyormuş gibi, paylaşılan değişkenlere erişme ve kritik bölgeler için atomikliği destekleme ihtiyacı gibi kurmuştur. Görünüşe göre, bir iş parçacığının devam etmeden önce bir eylemi tamamlaması için diğerini beklemesi gereken başka bir ortak etkileşim daha var. Bu karşılıklı eylem, örneğin, bir işlem bir disk I/O gerçekleştirdiğinde ve uyku moduna geçirdiğinde ortaya çıkar; I/O tamamlandıında, işlemin devam edebilmesi için uykusundan uyandırılması gereklidir.

Bu nedenle, öncümüzdeki bölgelerde, sadece atomikliği desteklemek için senkronizasyon ilkeleri için nasıl destek oluşturulacağını değil, aynı zamanda çok iş parçacıklı programlarda yaygın olan bu tür uyku / uyanık etkileşimi destekleyen mekanizmalar için nasıl destek oluşturulacağını da inceleyeceğiz. Bu şu anda mantıklı değilse, sorun değil! Yakında kondisyon değişkenleri ile ilgili bölüm okuduğunuzda yeterli olacaktır. O zamana kadar olmazsa, o zaman daha az sorun değil ve mantıklı olana kadar bu bölüm tekrar (ve tekrar) okumalısınız..

BIR KENARA: TEMEL EŞZAMANLILIK TERİMLERİ

KRITİK BÖLÜM, YARIŞ DURUMU, BELIRSİZ, KARŞILIKLI DİŞLAMA

Bu dört terim, eşzamanlı kod için o kadar merkezidir ki, onları açıkça belirtmenin zaman alacağını düşündük. Daha fazla ayrıntı için Dijkstra'nın erken dönem çalışmalarından bazılarına bakın [D65,D68].

- **Kritik bölüm(critical section)**, genellikle bir değişken veya veri yapısı olan paylaşılan bir kaynağa erişen bir kod parçasıdır.
- **Bir yarış durumu(race condition)** (veya **veri yarışı(data race)** [NM92]), birden fazla yürütme iş parçacığının kritik bölüme kabaca aynı anda girmesi durumunda ortaya çıkar; her ikisi de paylaşılan veri yapısını güncellemeye çalışır ve şartsız (ve belki de istenmeyen) bir sonuca yol açar.
- **Belirsiz(indeterminate)** bir program bir veya daha fazla yarış koşullarından oluşur; programın çıktıtı, hangi iş parçacıklarının ne zaman çalıştırıldığına bağlı olarak çalıştırmadan çalıştırılmaya取决于. Bu nedenle sonuç, genellikle bilgisayar sistemlerinden beklediğimiz bir şey olan **deterministik(deterministic)** değildir.
- Bu sorunlardan kaçınmak için, iplikler bir tür **karşılıklı dışlama(mutual exclusion)** ilkeleri kullanmalıdır; bunu yapmak, yalnızca tek bir iplığın kritik bir bölüme girmesini garanti eder, böylece yarışlardan kaçınır ve deterministik program çıktılarıyla sonuçlanır.

26.7 Özet: Neden OS Sınıfında?

Bitirmeden önce, aklınıza gelebilecek bir soru şudur: Bunu neden işletim sistemi sınıfında inceliyoruz? "Tarih" tek kelimeyle cevaptır; işletim sistemi ilk eşzamanlı programdı ve işletim sistemi içinde kullanılmak üzere birçok teknik oluşturuldu. Daha sonra, çok iş parçacıklı süreçlerle, uygulama programcıları da bu tür şeyleri düşünmek zorunda kaldılar.

Örneğin, çalışan iki işlemin olduğu durumu düşünün. İkisinin de aradığını varsayılm `write()` dosyaya yazmak, ve her ikisi de verileri dosyaya eklemek ister (yani, verileri dosyanın sonuna ekleyin, böylece uzunluğunu artırın). Bunu yapmak için, her ikisi de yeni bir blok ayırmalı, bu bloğun bulunduğu dosyanın `inode`'una kaydetmeli ve dosyanın boyutunu yeni daha büyük boyutu yansıtacak şekilde değiştirmelidir (diğer şeylerin yanı sıra; kitabı üçüncü bölümünde dosyalar hakkında daha fazla bilgi edineceğiz). Herhangi bir zamanda bir kesinti meydana gelebileceğinden, bu paylaşılan yapıları güncelleyen kod (örneğin, tahsis için bir bitmap veya dosyanın `inode`'u) kritik bölümlerdir; Bu nedenle, işletim sistemi tasarımcıları, kesmenin başlamasının en başından beri, işletim sisteminin iç yapıları nasıl güncellediği konusunda endişelenmek zorunda kaldılar. Zamansız bir ara dönem, yukarıda açıklanan tüm sorumlara neden olur. Şartsız olmayan bir şekilde, sayfa tabloları, işlem listeleri, dosya sistemi yapıları ve hemen hemen her çekirdek veri yapısına, doğru şekilde çalışması için uygun senkronizasyon ilkelleriley

dikkatlice erişilmesi gereklidir.

References

- [D65] “Solution of a problem in concurrent programming control” by E. W. Dijkstra. Communications of the ACM, 8(9):569, September 1965. *Pointed to as the first paper of Dijkstra’s where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.*
- [D68] “Cooperating sequential processes” by Edsger W. Dijkstra. 1968. Available at this site: <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF>. *Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Technische Hogeschool Eindhoven (THE), including this famous paper on “cooperating sequential processes”, which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the “THE” operating system (said “T”, “H”, “E”, and not like the word “the”).*
- [GR92] “Transaction Processing: Concepts and Techniques” by Jim Gray and Andreas Reuter. Morgan Kaufmann, September 1992. *This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray’s “brain dump”, in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.*
- [L+93] “Atomic Transactions” by Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete. Morgan Kaufmann, August 1993. *A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.*
- [NM92] “What Are Race Conditions? Some Issues and Formalizations” by Robert H. B. Netzer and Barton P. Miller. ACM Letters on Programming Languages and Systems, Volume 1:1, March 1992. *An excellent discussion of the different types of races found in concurrent programs. In this chapter (and the next few), we focus on data races, but later we will broaden to discuss general races as well.*
- [SR05] “Advanced Programming in the UNIX Environment” by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. *As we’ve said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.*

Ödev (Simülasyon)

x86.py bu program, farklı iplik ara ayrılmalarının yarış koşullarına nasıl neden olduğunu veya önlediğini görmenizi sağlar. Programın nasıl çalıştığını dair details için README'ye bakın, ardından aşağıdaki soruları cevaplayın.

Soru

- 1 . Basit bir program olan "loop.s"yi inceleyelim. İlk olarak, sadece okuyun ve derstandı açın. Ardından, bu bağımsız değişkenlerle çalıştırın (./x86.py -p loop.s -t 1 -l 100 -R dx) Bu, tek bir iş parçacığı, her 100 yönargedede bir kesme ve %dx kaydının izlenmesini belirtir. Koşu sırasında %dx ne olacak? Yanıtlarınızı kontrol etmek için -c bayrağını kullanın; soldaki answers, sağdaki talimat çalıştırıktan sonra kaydın değerini (veya bellek değerini) gösterir.

```
? 1000 sub $1,%dx  
? 1001 test $0,%dx  
? 1002 jgte .top
```

```
-1 1000 sub $1,%dx  
-1 1001 test $0,%dx  
-1 1002 jgte .top  
-1 1003 halt
```

- 2 . Aynı kod, farklı bayraklar: (./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx) Bu, iki iş parçacığı belirtir ve her %dx'li 3'e başlatır. %dx hangi değerleri görecek? Kontrol etmek için -c ile çalıştırın. Birden fazla iş parçacığının varlığı hesaplamalarınızı etkiliyor mu? Bu kodda bir yarış var mı?

```
?           1000 sub $1,%dx  
?           1001 test $0,%dx  
?           1002 jgte .top  
?           1003 halt
```

```
-1           1000 sub $1,%dx  
-1           1001 test $0,%dx  
-1           1002 jgte .top  
-1           1003 halt
```

- 3 . Bunu çalıştırın: ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx Bu, kesme aralığını küçük/rastgele yapar; farklı ara geçişleri görmek için farklı tohumlar (-s) kullanın. Kesme frekansı herhangi bir şeyi değiştirir mi?

CONCURRENCY: AN INTRODUCTION

21

```
$ python3 ./x86.py -p loop.s -t 2 -l 3 -r -a dx=3, dx=3 -R dx -s4
ARG seed 4
ARG numThreads 2
ARG programLoop.s
ARG interruptFrequency 3
ARG interruptRandomness True
ARG regDw 0x0
ARG memAddress 1000
ARG memSize 128
ARG memTrace
ARG registerTrace dx
ARG cctrace False
ARG printStats False
ARG verbose False

dx      Thread 0      Thread 1
?
? 1000 sub $1,%dx
? 1001 test $0,%dx
? 1002 jgte .top
? .... Interrupt ..... Interrupt .....
? 1000 sub $1,%dx
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1000 sub $1,%dx
? 1001 test $0,%dx
? .... Interrupt ..... Interrupt .....
? 1002 jgtn .top
? .... Halt;Switch ..... Halt;Switch .....
? .... Interrupt ..... Interrupt .....
? 1002 jgtn .top
? 1000 sub $1,%dx
? .... Interrupt ..... Interrupt .....
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1003 halt

abduselam@abduselam-virtual-machine:~/Desktop/osstep-homework-master/threads/intro$ python3 ./x86.py -p loop.s -t 2 -l 3 -r -a dx=3, dx=3 -R dx -s4
ARG seed 5
ARG numThreads 2
ARG programLoop.s
ARG interruptFrequency 3
ARG interruptRandomness True
ARG regDw 0x0
ARG memAddress 1000
ARG memSize 128
ARG memTrace
ARG registerTrace dx
ARG cctrace False
ARG printStats False
ARG verbose False

dx      Thread 0      Thread 1
?
? 1000 sub $1,%dx
? .... Interrupt ..... Interrupt .....
? .... Interrupt ..... Interrupt .....
? 1000 sub $1,%dx
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1003 halt

abduselam@abduselam-virtual-machine:~/Desktop/osstep-homework-master/threads/intro$ python3 ./x86.py -p loop.s -t 2 -l 3 -r -a dx=3, dx=3 -R dx -s5
ARG seed 5
ARG numThreads 2
ARG programLoop.s
ARG interruptFrequency 3
ARG interruptRandomness True
ARG regDw 0x0
ARG memAddress 1000
ARG memSize 128
ARG memTrace
ARG registerTrace dx
ARG cctrace False
ARG printStats False
ARG verbose False

dx      Thread 0      Thread 1
?
? 1000 sub $1,%dx
? 1001 test $0,%dx
? .... Interrupt ..... Interrupt .....
? 1000 sub $1,%dx
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1003 halt

.... Halt;Switch ..... Halt;Switch .....
? 1002 jgtn .top
? 1000 sub $1,%dx
? .... Interrupt ..... Interrupt .....
? 1001 test $0,%dx
? 1002 jgtn .top
? .... Interrupt ..... Interrupt .....
? 1003 halt
```

- 4 . Şimdi, 2000 adresinde bulunan paylaşılan bir değişkeni destekleyen looping-race-nolock.s adlı farklı bir program; buna değişken değer diyeceğiz. Ayakta durmanızı doğrulamak için tek bir iş parçacığıyla çalıştırın: ./x86.py -p looping-race-nolock.s -t 1 -M 2000 Koşu boyunca değer nedir (yani, bellek adresi 2000'de)? Kontrol etmek için -c kullanın.

```

6          1004 test $0, %bx
6          1005 jgt .top
6          1006 halt
?
?          1004 test $0, %bx
?          1005 jgt .top
?          1006 halt
?
?          1004 test $0, %bx
?          1005 jgt .top
?          1006 halt
?
1 1004 test $0, %bx
1 1005 jgt .top
1 1006 halt

```

- 5 . Birden çok yineleme/iş parçacığı ile çalıştırın: ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000 Neden her iş parçacığı üç kez döngü yapıyor? Değerin nihai değeri nedir?

Çünkü bx, üç olarak başlatılmıştır.

Altı

6. Rastgele kesme aralıklarıyla çalıştırın: ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0 farklı tohumlarla (-s 1, -s 2, vb.) Değerin nihai değerinin ne olacağını birbirine bağlayan iş parçacığına bakarak söyleyebilir misiniz? Kesintinin zamanlaması önemli mi? Nerede güvenli bir şekilde oluşabilir? Başka bir deyişle, kritik bölüm tam olarak nerede?

Yükseltilmiş baltanın sıfırlanmadan önce "değere(value)" kaydedilip kaydedilmemiği.

```
1 ----- Interrupt ----- Interrupt -----  
2 1006 halt  
3 ----- Halt;Switch ----- Halt;Switch -----  
4  
5 ----- Halt;Switch ----- Halt;Switch -----  
6 1006 halt  
7
```

```
1 ----- Interrupt ----- Interrupt -----  
2 1006 halt  
3 ----- Halt;Switch ----- Halt;Switch -----  
4  
5 ----- Halt;Switch ----- Halt;Switch -----  
6 1006 halt  
7 ----- Interrupt ----- Interrupt -----  
8 1006 halt  
9
```

```
1 1006 halt  
2 ----- Halt;Switch ----- Halt;Switch -----  
3 ----- Interrupt ----- Interrupt -----  
4 1006 halt  
5
```

7. Şimdi sabit kesme aralıklarını inceleyin: ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1 Paylaşılan değişken değerinin nihai değeri ne olacak? Peki ya -i 2, -i 3 vb. değiştirdiğinizde? Program hangi kesinti aralıkları için "doğru" cevabı veriyor?

```
abduselam@abduselam-Virtual-Machine:~/Desktop/ostep-homework-master/threads-intro$ python3 ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 1
ARG seed 0
ARG numthreads 2
ARG program ./looping-race-nolock.s
ARG interrupt_frequency 4
ARG interrupt_randomness True
ARG argv
ARG load_address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0      Thread 1
?
? 1000 mov 2000, %ax
? 1001 add $1, %ax
? 1002 mov %ax, 2000
? 1003 sub $1, %bx
? ----- Interrupt -----
? 1004 mov 2000, %ax
? 1005 add $1, %ax
? 1006 mov %ax, 2000
? 1007 sub $1, %bx
? ----- Interrupt ----- Interrupt -----
? 1008 test $0, %bx
? 1009 jgt .top
? ----- Interrupt -----
? 1010 test $0, %bx
? 1011 jgt .top
? ----- Interrupt ----- Interrupt -----
? 1012 halt
? ----- Halt;Switch -----
? 1006 halt
?
```

```
abduselam@abduselam-Virtual-Machine:~/Desktop/ostep-homework-threads-intro$ python3 ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 2
ARG seed 0
ARG numthreads 2
ARG program ./looping-race-nolock.s
ARG interrupt_frequency 1
ARG interrupt_randomness False
ARG argv
ARG load_address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0      Thread 1
?
? 1000 mov 2000, %ax
? ----- Interrupt -----
? 1001 add $1, %ax
? ----- Interrupt -----
? 1002 mov %ax, 2000
? 1003 sub $1, %bx
? ----- Interrupt -----
? 1004 add $1, %ax
? ----- Interrupt -----
? 1005 mov %ax, 2000
? 1006 sub $1, %bx
? ----- Interrupt -----
? 1007 test $0, %bx
? ----- Interrupt -----
? 1008 jgt .top
? ----- Interrupt -----
? 1009 test $0, %bx
? ----- Interrupt -----
? 1010 jgt .top
? ----- Interrupt -----
? 1011 halt
? ----- Halt;Switch -----
? 1006 halt
?
```

```

ARG seed 0
ARG numthreads 2
ARG program ./looping-race-nolock.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG argb load 0
ARG memsize 1000
ARG memtrace 2000
ARG verbose True
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0           Thread 1
0         ..... Interrupt .....
0 1000 mov $0, %ax
0 1000 mov 2000, %ax
0 1000 mov 2000, %ax
0 1000 mov 2000, %ax
0 1001 add $1, %ax
0 1001 add $1, %ax
0 1001 add $1, %ax
0 1001 add $1, %ax
0 1002 mov %ax, 2000
0 1002 mov %ax, 2000
0 1002 mov %ax, 2000
0 1002 mov %ax, 2000
0 1003 sub $1, %bx
0 1003 sub $1, %bx
0 1003 sub $1, %bx
0 1003 sub $1, %bx
0 1004 test $0, %bx
0 1004 test $0, %bx
0 1004 test $0, %bx
0 1004 test $0, %bx
0 1005 jgt .top
0 1005 jgt .top
0 1006 halt
1 1006 Halt;Switch .....
1 1006 Halt;Switch .....
1 1006 halt
1 1006 halt

#ัดดิสแลน@desktop-an-virtual-machine:~/Desktop/ostep-homework-master/threads-intro$ python3 ./x86.py -p looping-race-nolock.s -a bx=1 -t2 -M 2000 -l 2
ARG seed 0
ARG numthreads 2
ARG program ./looping-race-nolock.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG argb load address 1000
ARG memsize 128
ARG memtrace 2000
ARG verbose True
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0           Thread 1
7         ..... Interrupt .....
7 1000 mov 2000, %ax
7 1001 add $1, %ax
7 1001 add $1, %ax
7 1001 add $1, %ax
7 1002 mov %ax, 2000
7 1002 mov %ax, 2000
7 1002 mov %ax, 2000
7 1002 mov %ax, 2000
7 1003 sub $1, %bx
7 1003 sub $1, %bx
7 1003 sub $1, %bx
7 1003 sub $1, %bx
7 1004 test $0, %bx
7 1004 test $0, %bx
7 1004 test $0, %bx
7 1004 test $0, %bx
7 1005 jgt .top
7 1005 jgt .top
7 1006 halt
7 1006 Halt;Switch .....
7 1006 halt

#ัดดิสแลน@desktop-an-virtual-machine:~/Desktop/ostep-homework-master/threads-intro$ python3 ./x86.py -p looping-race-nolock.s -a bx=1 -t2 -M 2000 -l 2 -c
ARG seed 0
ARG numthreads 2
ARG program ./looping-race-nolock.s
ARG interrupt frequency 2
ARG interrupt randomness False
ARG argb load address 1000
ARG memsize 128
ARG memtrace 2000
ARG retrace True
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0           Thread 1
0         ..... Interrupt .....
0 1000 mov $1, %ax
0 1001 add $1, %ax
0 1001 add $1, %ax
0 1001 add $1, %ax
0 1002 mov $0, 2000
0 1003 sub $1, %bx
1 1004 test $0, %bx
1 1004 test $0, %bx
1 1004 test $0, %bx
1 1004 test $0, %bx
1 1005 jgt .top
1 1005 jgt .top
1 1006 halt
1 1006 Halt;Switch .....
1 1006 halt

```

```

ARG seed 0
ARG numthreads 2
ARG program ./x86.py -p looping-race-nolock.s
ARG interrupt frequency 3
ARG interrupt randomness False
ARG argv bx1
ARG load address 1000
ARG memsize 128
ARG memory size 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0      Thread 1
?
? 1000 mov 2000, %ax
? 1001 add $1, %ax
? 1002 mov %ax, 2000
?     ; Interrupt -----
?     ;----- Interrupt -----
? 1000 mov 2000, %ax
? 1001 add $1, %ax
? 1002 mov %ax, 2000
?     ;----- Interrupt -----
? 1003 sub $1, %bx
? 1004 test $0, %bx
? 1005 jgt .top
?     ;----- Interrupt -----
? 1003 sub $1, %bx
? 1004 test $0, %bx
? 1005 jne .top
?     ;----- Interrupt -----
? 1006 halt
?     ;----- Halt;Switch -----
? 1006 halt

address@localhost:~/Desktop/ostep-homework-master/thread-intro$ python3 ./x86.py -p looping-race-nolock.s -a bx=1 -t2 -M 2000 -l 3 -c

ARG seed 0
ARG numthreads 2
ARG program looping-race-nolock.s
ARG interrupt frequency 3
ARG interrupt randomness False
ARG argv bx1
ARG load address 1000
ARG memsize 128
ARG memory size 2000
ARG retrace
ARG cctrace False
ARG printstats False
ARG verbose False

2000      Thread 0      Thread 1
0 1000 mov 2000, %ax
0 1001 add $1, %ax
1 1002 mov %ax, 2000
1     ;----- Interrupt -----
1     ;----- Interrupt -----
1 1000 mov 2000, %ax
1 1001 add $1, %ax
2 1002 mov %ax, 2000
2     ;----- Interrupt -----
2 1003 sub $1, %bx
2 1004 test $0, %bx
2 1005 jgt .top
2     ;----- Interrupt -----
2 1003 sub $1, %bx
2 1004 test $0, %bx
2 1005 jne .top
2     ;----- Interrupt -----
2 1006 halt
2     ;----- Halt;Switch -----
2 1006 halt

```

8. Daha fazla döngü için de aynısını çalıştırın (örneğin, set -a bx=100). Hangi aralık aralıkları (-i) doğru bir sonuca yol açar? Hangi aralıklar şaşırıcı?

Aralıklar(intervals) $\geq 597, ==3 * x$ ($x = 1,2,3\dots$)

9. Son bir program: wait-for-me.s. Çalıştır: ./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000 Bu, %ax kaydını iş parçacığı 0 için 1 ve iş parçacığı 1 için 0 olarak ayarlar ve %ax ve bellek konumu 2000'i izler. Kod nasıl davranışlıdır? Konum 2000'deki değer iş parçacıkları tarafından nasıl kullanılıyor? Nihai değeri ne olacak?

İş parçacığı 0, belleği 2000'i 1'e ayarlar ve ardından durur. İş parçacığı 1,

```
abdusselam@abdusselam-virtual-machine:~/Desktop/ostep-homework-Master/threads-intro$ python3 ./x86.py -p wait-for-me.s -a ax=1, ax=0 -R ax -M 2000
ARG seed 0
ARG numthreads 2
ARG program wait-for-me.s
ARG interrupt_frequency 50
ARG interrupt_randomness False
ARG argv ax=1
ARG load_address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False

2000      ax          Thread 0          Thread 1
?          ?          Thread 0          Thread 1
?          ? 1000 test $1, %ax
?          ? 1001 je .signaller
?          ? 1006 mov $1, 2000
?          ? 1007 halt
?          ?      ---- Halt;Switch ----
?          ?          1000 test $1, %ax
?          ?          1001 je .signaller
?          ?          1002 mov 2000, %cx
?          ?          1003 test $1, %cx
?          ?          1004 jne .waiter
?          ?          1005 halt
```

```
abdusselam@abdusselam-virtual-machine:~/Desktop/ostep-homework-Master/threads-intro$ python3 ./x86.py -p wait-for-me.s -a ax=1, ax=0 -R ax -M 2000 -c
ARG seed 0
ARG numthreads 2
ARG program wait-for-me.s
ARG interrupt_frequency 50
ARG interrupt_randomness False
ARG argv ax=1,
ARG load_address 1000
ARG memsize 128
ARG memtrace 2000
ARG regtrace ax
ARG cctrace False
ARG printstats False
ARG verbose False

2000      ax          Thread 0          Thread 1
0          1          Thread 0          Thread 1
0          1 1000 test $1, %ax
0          1 1001 je .signaller
1          1 1006 mov $1, 2000
1          1 1007 halt
1          0      ---- Halt;Switch ----
1          0          1000 test $1, %ax
1          0          1001 je .signaller
1          0          1002 mov 2000, %cx
1          0          1003 test $1, %cx
1          0          1004 jne .waiter
1          0          1005 halt
```

bellek 2000 1 olana kadar bir döngüde çalışmaya devam eder.

- Şimdi girişleri değiştirin: `./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000` İplikler nasıl davranışır? Thread 0 ne yapıyor? Kesme aralığını değiştirmek (örneğin, `-i 1000` veya belki de rastgele aralıklar kullanmak) izleme sonucunu

nasıl değiştirir? Program CPU'yu verimli bir şekilde kullanıyor mu?